Software quality as a software architecture driver

A microservices architecture example

09.06.2021

Agenda

1 Why does software quality matter?

2 What is it and how do we achieve it?

3 Enterprise software

4 Technologies for microservice implementation

5 Modern pipelines

6 Lessons learned

7 Concluding remarks

1.

Why does software quality matter?

1.01 We work with a modern pipeline

- Agile Methologies:
  - Short development (and release) cycles allowing fast adaptability to business needs
- Continuous integration
  - Build triggered on code commit
  - Automated tests running
  - Quality code checks (e.g., static code analyzer)
- Continuous Delivery
  - Release process automation
  - Faster, repeatable, reliable releases
- Cloud computing
  - Easy hardware provisioning
  - Easy scalability – use what you need, as much as you need, when you need it, at a reasonable cost

1.02. What do modern pipelines mean for software development?

- We have mechanisms that allow us:
  - Rapidly develop software, test it, build it and deploy it

  - Develop the software in short delivery cycles that adapt to the business needs, even when they change quickly:
    - Response to competition
    - Response to changing market need

  - Our software needs to be adaptive to change, too
    - This happens regardless of where we deploy it
    - The cloud just made possible accessing infrastructure without an upfront cost

  - So the big question is:
    - How do we achieve software that can be as adaptive to change as possible?

2.

What is software quality and do we achieve it?

2.01. Software quality attributes

- ISO 25010 defines 8 quality characteristics:
  - Functional suitability
  - Performance efficiency
  - Compatibility
  - Usability
  - Reliability
  - Security
  - **Maintainability**
  - Portability

2.02. Maintainability (1)

- Modularity
  - Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

- Reusability
  - Degree to which an asset can be used in more than one system, or in building other assets.

- Analyzability
  - Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.

2.03. Maintainability (2)

- Modifiability
  - Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.

- Testability
  - Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component
  - Degree to which tests can be performed to determine whether those criteria have been met.

2.04. How do we achieve maintainability?

- Design guidelines deemed important (by industry practitioners):
  - Coupling (want it low)
    - Connections between components / classes
  - Cohesion (want it high)
    - How related are the components / classes
  - Complexity (the higher the worse)
  - Size (the higher the worse)
  - **Programming to an interface**

2.05. How do we achieve maintainability?

- Design guidelines deemed important (by industry practitioners):
  - Coupling (want it low)
    - Connections between components / classes
  - Cohesion (want it high)
    - How related are the components / classes
  - Complexity (the higher the worse)
  - Size (the higher the worse)
  - **Programming to an interface**

2.06. SOLID Principles

- 5 Principles:
  - Single Responsibility Principle - SRP
  - Open Closed Principle – ORP
  - Liskov Substitution Principle – LSP
  - Interface Segregation Principle – ISP
  - Dependency Inversion Principle – DIP

2.07. Single Responsibility Principle

- The principle:
  - **One class must have one and only one reason to change**
  - If one class has more than one reason to change, it has more than one responsibility, it needs to be decomposed into smaller classes
  - Each of the resulting classes must have one reason to change, thus a single responsibility itself or else it should be decomposed
  - **We can apply a similar reasoning to a method (at a lower level) or even a module (at a higher level) – they should have a single responsibility**

2.08. Single Responsibility Principle / Example

- The method does 3 things:
  - Send the invitation
  - Validate the name
  - Validate the email
- We have, in fact, 3 reasons for the method to change
- If we had 3 methods in the class (SendInvitation, ValidateName, ValidateEmail), they would not be cohesive, either

This class does not respect the SRP

```
public class InvitationService
   {
      public void SendInvite(string email, string firstName, string
lastName)
      {
         if (String.IsNullOrWhiteSpace(firstName) ||
String.IsNullOrWhiteSpace(lastName))
         {
            throw new Exception("Name is not valid!");
         }
         if (!email.Contains("@") || !email.Contains("."))
         {
            throw new Exception("Email is not valid!!");
         }
         SmtpClient client = new SmtpClient();
         client.Send(new MailMessage("mysite@nowhere.com", email) {
Subject = "Please join me at my party!" });
      }
   }
```

2.09. Single Responsibility Principle / Solution (1)

```
public class EmailService
   {
      public void Validate(string email)
      {
         if (!email.Contains("@") || !email.Contains("."))
         {
            throw new Exception("Email is not valid!!");
         }
      }
   }
```

```
public class UserNameService
   {
      public void Validate(string firstName, string
lastName)
      {
         if (String.IsNullOrWhiteSpace(firstName) ||
String.IsNullOrWhiteSpace(lastName))
         {
            throw new Exception("The name is invalid!");
         }
      }
   }
```

2.10. Single Responsibility Principle / Solution (2)

```
public class NewInvitationService
   {
      UserNameService _userNameService;
      EmailService _emailService;

      public NewInvitationService(UserNameService userNameService, EmailService emailService)
      {
         _userNameService = userNameService;
         _emailService = emailService;
      }
      public void SendInvite(string email, string firstName, string lastName)
      {
         _userNameService.Validate(firstName, lastName);
         _emailService.Validate(email);
         SmtpClient client = new SmtpClient();
         client.Send(new MailMessage("sitename@invites2you.com", email) { Subject = "Please join me at my party!" });
      }
   }
```

2.11. Single Responsibility Principle

- Primary benefit of applying the SRP:
  - **High cohesion**

- If one class has a single responsibility, its methods will be related to a common, underlying subject, thus ensuring high cohesion

- Do we need the SRP?
  - If we remind ourselves of making sure our classes are cohesive, not really

  - We should keep it in mind though

  - Some people have one more general rule:
    - Write simple code
    - This may be too generic, though

2.12. The Dependency Inversion Principle

- The principle:
  - **High-level modules should not depend on low-level modules. Both should depend on abstractions.**
  - **Abstractions should not depend on details. Details should depend on abstractions.**

2.13. The Dependency Inversion Principle - Example

```
public class Email
   {

      public string ToAddress { get; set; }
      public string Subject { get; set; }
      public string Content { get; set; }
      public void SendEmail()
      {
         //Send email
      }
   }
```

```
public class SMS
   {
      public string PhoneNumber { get;
set; }
      public string Message { get; set; }
      public void SendSMS()
      {
         //Send sms
      }
   }
```

```
public class Notification
   {
      private Email _email;
      private SMS _sms;
      public Notification()
      {
         _email = new Email();
         _sms = new SMS();
      }

      public void Send()
      {
         _email.SendEmail();
         _sms.SendSMS();
      }
   }
}
```

- Notification (higher level) depends on
  - Email (lower level)
  - SMS (lower level)

- **We have a violation of the DIP**
  - Notification depends on concrete, lower level classes
  - We also have high coupling between Notification and Email and SMS

2.14. The Dependency Inversion Principle – Example (2)

```
public interface IMessage
   {
      void SendMessage();
   }
```

```
public class Email : IMessage
   {

      public string ToAddress { get; set; }
      public string Subject { get; set; }
      public string Content { get; set; }
      public void SendMessage()
      {
         //Send email
      }
   }
```

```
public class SMS: IMessage
   {
      public string PhoneNumber { get;
set; }
      public string Message { get; set; }
      public void SendMessage()
      {
         //Send sms
      }
   }
```

```
public class NewNotification
   {
      private ICollection<IMessage> _messages;

      public
NewNotification(ICollection<IMessage>
messages)
      {
         this._messages = messages;
      }
      public void Send()
      {
         foreach (var message in _messages)
         {
            message.SendMessage();
         }
      }
   }
```

2.16. The Dependency Inversion Principle

- The NewNotification class does not rely on details, on concretions
  - It relies on abstractions: interfaces
  - By doing this, we are respecting the DIP
- Primary benefit of applying the DIP:
  - Low coupling
- By depending on abstractions and not concrete classes, we are lowering the coupling in our code

2.17. The Open / Closed Principle

- The principle:
  - **A given software entity should be open for extension, but closed for modification**
  - To be open for extension developers must be able to respond to changing requirements and support new features.
  - This must be achieved despite modules being closed to modification.
  - Developers must support new functionality without editing the source code
  - Can be achieved in multiple ways, but interfaces can have a role

2.18. The Liskov Substitution Principle

- The principle:
  - **We should be able to treat a child class as though it were the parent class**
  - Classes that inherit from a given class should not change the functionality of the parent class
  - This principle tells us how to build inheritance hierarchies where all the types can be treated similarly

2.19. The Interface Segregation Principle

- The principle:
  - **No client code object should be forced to depend on methods it does not use**
- In practical terms
  - Interfaces should be as small as possible
    - Each method added is an additional obligation for an implementer
  - We should consider splitting them, if they are not small or simple

2.20. Summary

- **Write adaptive, easily modifiable code**
- Use of interfaces
- Single Responsibility Principle
- Write simple code

3.

Microservices

3.01. The reality of enterprise software

- Complex, legacy software
- Hard to change (complex monoliths)
- May be relatively complex to deploy
- Poor test coverage
- Hard to make it benefit from the advantages of the cloud
- Several architectural attempts at solving these problems (SOA), not entirely successful
- Microservices are another attempt at solving these problems

3.02. Microservices goals

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A microservice consists of one or more processes.
- A microservice owns its own data store.
- A small team can maintain a few handfuls of microservices.
- **A microservice is (easily) replaceable**.

3.03. Microservices are / should be more maintainable

- Well-factored and well-implemented microservices are highly maintainable from a couple of perspectives.
- From a developer perspective, several factors play a part in making microservices maintainable:
  - Each well-factored microservice provides a single capability.
  - A microservice owns its own data store. No other services can interfere with a microservice's data store.
    - This, combined with the typical size of the codebase for a microservice, means you can understand a complete service all at once.
- Well-written microservices can (and should) be comprehensibly covered by automated tests

3.04. Microservices should be lightweight

- Because every microservice handles a single capability, microservices are by nature fairly small both in their scope and in the size of their codebase.
  - The simplicity that follows from this limited scope is a major benefit of microservices.

  - Avoid complicating their codebase by using large, complicated frameworks, libraries, or products
    - prefer smaller, lightweight technologies that do what the microservice needs right now

  - Remember, a microservice is replaceable; it can be completely rewritten within a reasonable budget if at some point the technologies you used originally no longer meet your needs

3.05. Is there a size limit for a microservice?

- When developing a microservice, size should not be the important point
- Instead, the important point should be to create loosely coupled services, so you have autonomy of development, deployment, and scale, for each service.
- When identifying and designing microservices, you should try to make them as small as possible
  - Avoid, as much as possible, direct dependencies with other microservices.
- More important than the size of the microservice is the **internal cohesion** it must have and its independence from other services
  - (Cohesion, SRP)
  - Cohesion is a way to identify how to break apart or group together microservices

3.07. Organizing microservices for use by different applications – the API Gateway

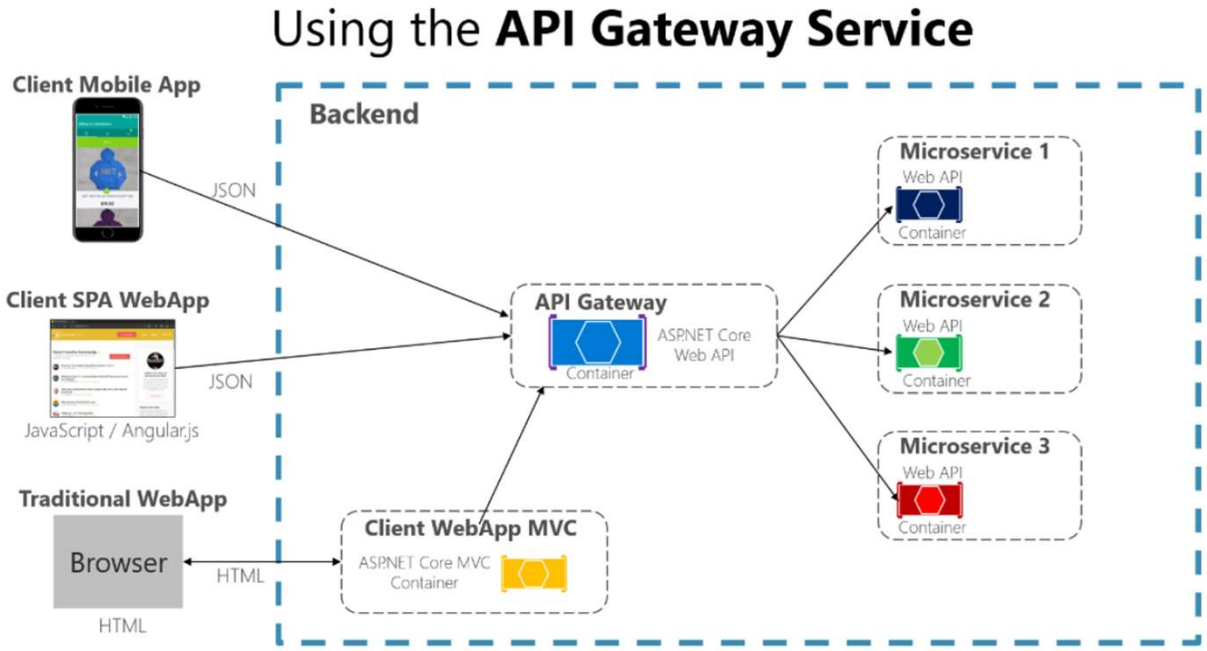


***Figure 4-13**. Using an API Gateway implemented as a custom Web API service*

3.06. Organizing microservices for use by different applications – the API Gateway

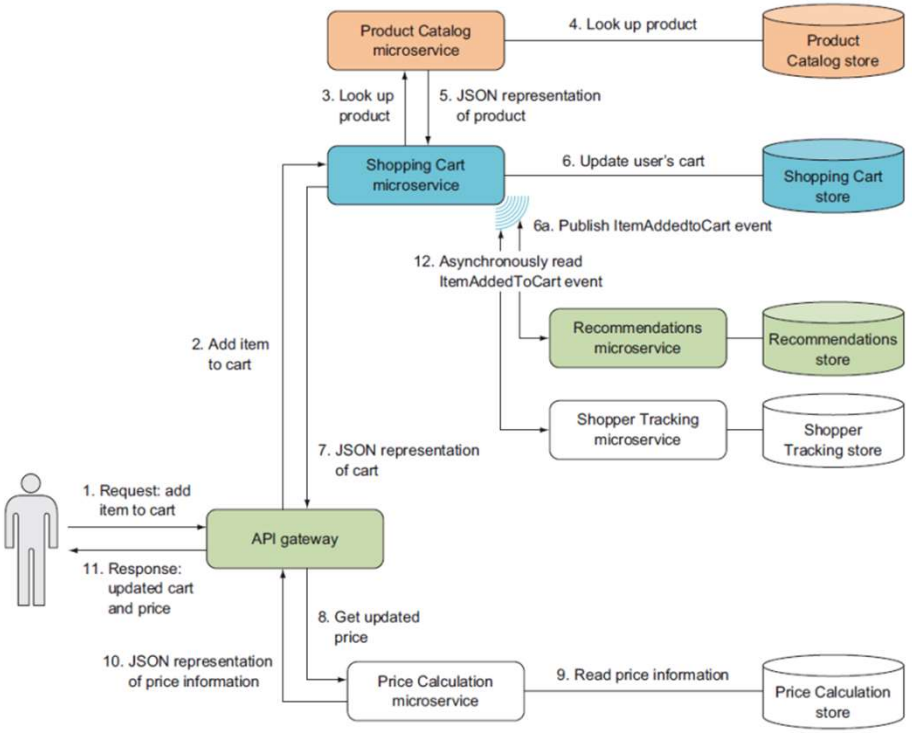


Figure 1.8 When a user adds an item to their shopping cart, the front end makes a request to the API Gateway microservice, which collaborates with other microservices to fulfill the request. During processing, microservices may raise events that other microservices can subscribe to and handle asynchronously.

3.07. One or more gateways?

- Probably several API Gateways should be implemented, so that you can have a different façade for the needs of each client app.
- Each API Gateway can provide a different API tailored for each client app, possibly even based on the client form factor by implementing specific adapter code which underneath calls multiple internal microservices.
- Since a custom API Gateway is usually a data aggregator, you need to be careful with it.
  - Usually it isn't a good idea to have a single API Gateway aggregating all the internal microservices of your application.
  - If it does, it acts as a monolithic aggregator or orchestrator and violates microservice autonomy by coupling all the microservices.
- Therefore, the API Gateways should be segregated based on business boundaries and not act as an aggregator for the whole application

4.

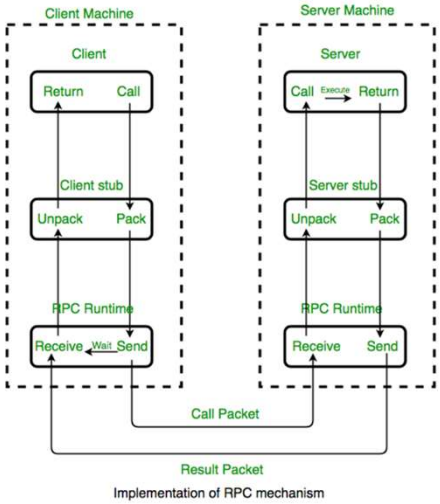
Technologies for microservices implementation

4.01. Technologies

- Microservices can be a bit like a class, in the sense that they should respect the SRP – one single responsibility
- Through high cohesion and low coupling, we can build a system that is “easy” to develop and maintain
- When we think about this type of architecture, we can see quality characteristics exhibited at two levels:
  - Choosing microservices scope (Cohesion)
  - Microservices implementation (Cohesion, Low coupling, up to the ownership of the database)
  - The languages and platforms for implementation can vary, can be mixed, and are not that relevant
    - I have worked in teams that used .Net core, Javascript / node.js, but there are other languages / platforms that can be used

4.02. Synchronous communication with and between microservices

- REST:
  - Well known paradigm based in technologies that power the internet we all know and use (HTTP, JSON, etc)
- gRPC
  - An increasingly popular open source, high performance, lightweight, remote procedure call framework
    - HTTP/2
    - Protocol buffers is used as the interface description language
    - Language agnostic
    - Contract first
- Both used in synchronous scenarios
  - Client expects and obtains an immediate response from the server



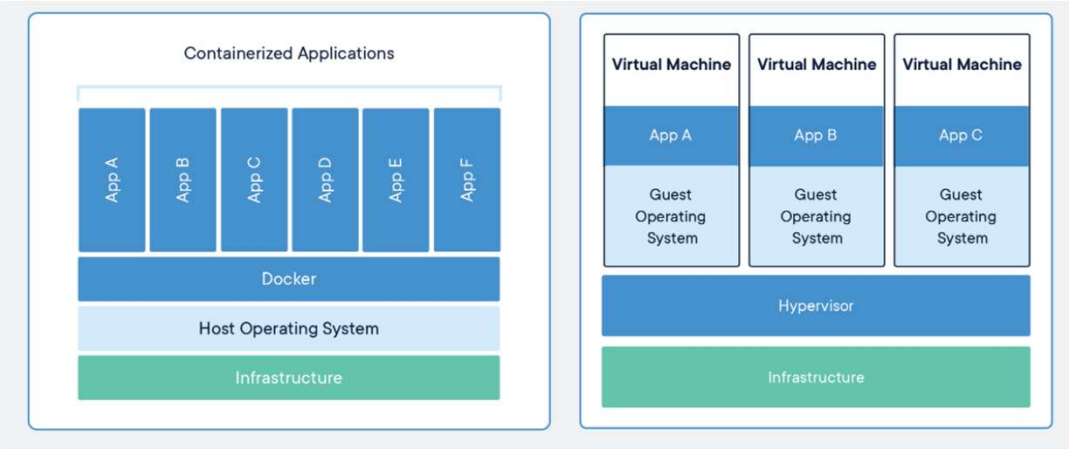
Implementation of RPC mechanism

4.04. Asynchronous communication: Message broker

- What is a message broker?
  - It allows communication based on exchange of messages allowing for very low coupling and very high scalability
    - Sending **commands** (to a known instance, direct communication)
    - Sending **events** describing the occurrence of a something relevant (events) to which all interested parties can subscribe
  - Delivery assurance (usually), even when target is offline
  - Raises its own difficulties
    - Testing included

4.05. Containers (Docker)

- A container packages up code and all its dependencies
- An application runs quickly and reliably from one computing environment to another.
- A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.
- Multiple containers can run on the same machine
- Share OS kernel
- Are more lightweight than Virtual Machines



4.05. Container management - Kubernetes

- Managing containers:
  - Deployment, scaling and management of containerized applications
    - Ensuring microservices are alive and well
    - Start new containers if needed
    - Destroying containers, when they are no longer needed
  - When containers are used, Kubernetes has a major role in scaling services
    - It’s a lot easier and faster to manage container lifecycle than to manage servers

5.

Modern pipelines

ALTRAN

5.01. Pipelines

- Modern development usually includes a pipeline where several tasks are performed:
  - Build
  - Run tests (unit and integration)
  - Provision hardware (servers, database servers) – infrastructure as code
  - Configure environments (test / stage/ production)
  - Container startup and management (single host / Kubernetes)

5.03. Pipeline configuration and management

- Domain of DevOps
  - A critical area, today
  - Specific languages for several purposes (e.g. terraform)
  - Frequently configured using a YAML file

Commit: A code change.

Job: Instructions that a runner has to execute.

Pipeline: A collection of jobs split into different stages.

Runner: An agent or server that executes each job individually that can spin up or down as needed.

Stages: A keyword that defines certain stages of a job, such as `build` and `deploy`. Jobs of the same stage are executed in parallel.

6.

Lessons learned

6.01. Lessons learned

- DevOps is a critical area that can seriously impact a team, when the competence is lacking
- Testing is critical for software development, but even more so for scenarios where CI / CD are present
- Knowing principles is always relevant, but the more complex the technology, the more relevant their knowledge is
- Preparation and vision matter
  - An organization needs to know where it wants to go and should prepare as extensively as possible to get here

7.

Concluding remarks

7.01. Concluding remarks

- Everything we do has a reason behind it
  - Whenever possible we should know those reasons
  - Software development, no matter how fancy a new technology, is always about the same goals and the underlying quality characteristics we look for, really do not change
  - As always, it is quite an interesting time to work on IT and software development
  - Take advantage of that